7 Automating operational tasks with Lambda

This chapter covers

- Creating a Lambda function to perform periodic health checks
- Triggering a Lambda function with CloudWatch events
- Searching through your Lambda function's logs with CloudWatch
- Monitoring Lambda functions with CloudWatch alarms
- Configuring IAM roles so Lambda functions can access other services
- Web application, data processing, and IoT with AWS Lambda
- Limitations of AWS Lambda

This chapter is about adding a new tool to your toolbox. The tool we're talking about, AWS Lambda, is as flexible as a Swiss Army Knife. You don't need a virtual machine to run your own code anymore, as AWS Lambda offers execution environments for Java, Node.js, C#, Python, and Go. All you have to do is to implement a

function, upload your code, and configure the execution environment. Afterward, your code is executed within a fully managed computing environment. AWS Lambda is well-integrated with all parts of AWS, enabling you to easily automate operations tasks within your infrastructure. We use AWS to automate our infrastructure regularly. For example, we use it to add and remove instances to a container cluster based on a custom algorithm, and to process and analyze log files.

AWS Lambda offers a maintenance-free and highly available computing environment. You no longer need to install security updates, replace failed virtual machines, or manage remote access (such as SSH or RDP) for administrators. On top of that, AWS Lambda is billed by invocation. Therefore, you don't have to pay for idling resources that are waiting for work (for example, for a task triggered once a day).

In our first example, you will create a Lambda function that performs periodic health checks for your website. This will teach you how to use the Management Console and offer a blueprint for getting started with AWS Lambda quickly. In our second example, you will learn how to write your own Python code and deploy a Lambda function in an automated way using CloudFormation. Your Lambda function will automatically add a tag to newly launched EC2 instances. At the end of the chapter, we'll show you additional use cases like building web applications, Internet of Things (IoT) back ends, or processing data with AWS Lambda.

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

But what is AWS Lambda? Before diving into our first real-world example, we will start with a short introduction.

7.1 Executing your code with AWS Lambda

Computing capacity is available at different layers of abstraction on AWS: virtual machines, containers, and functions. You learned about the virtual machines offered by Amazon's EC2 service in chapter 3. Containers offer another layer of abstraction on top of virtual machines. We don't cover containers as it would go beyond the scope of our book. AWS Lambda provides computing power as well but in a fine-grained manner: an execution environment for small functions, rather than a full-blown operating system or container.

7.1.1 What is serverless?

When reading about AWS Lambda, you might have stumbled upon the term *serverless* The following quote summarizes the confusion created by a catchy and provocative phrase:

> *[...] the word serverless is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.*
>
> —Peter Sbarski, *Serverless Architectures on AWS* (Manning, 2017)

We define a serverless system as one that meets the following criteria:

- No need to manage and maintain virtual machines.
- Fully managed service offering scalability and high availability.
- Billed per request and by resource consumption.
- Invoke the function to execute your code in the cloud.

AWS is not the only provider offering a serverless platform. Google (Cloud Functions) and Microsoft (Azure Functions) are other competitors in this area.

7.1.2 Running your code on AWS Lambda

As shown in figure 7.1, to execute your code with AWS Lambda, follow these steps:

1. Write the code.
2. Upload your code and its dependencies (such as libraries or modules).
3. Create a function determining the runtime environment and configuration.
4. Invoke the function to execute your code in the cloud.

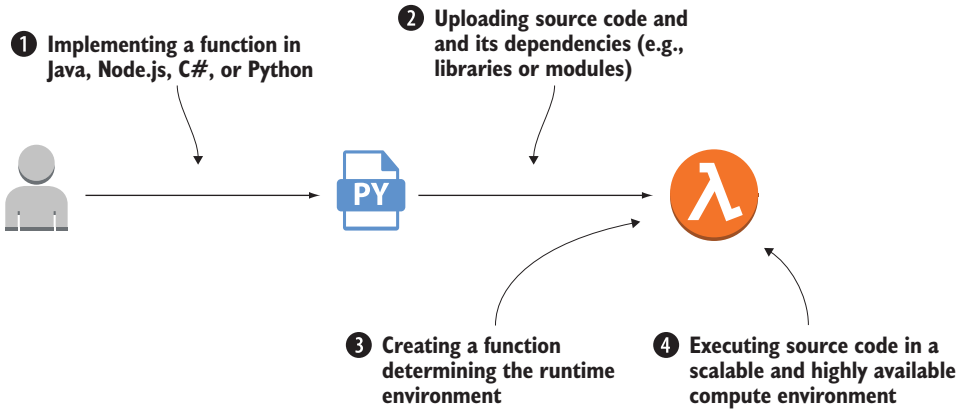


Figure 7.1 Executing code with AWS Lambda

You don't have to launch any virtual machines. AWS executes your code in a fully managed computing environment.

Currently, AWS Lambda offers runtime environments for the following languages:

- Java
- Node.js
- C#
- Python
- Go

Next, we will compare AWS Lambda with EC2 virtual machines.

7.1.3 Comparing AWS Lambda with virtual machines (Amazon EC2)

What is the difference between using AWS Lambda and virtual machines? First, there is the granularity of virtualization. Virtual machines provide a full operating system for running one or multiple applications. In contrast, AWS Lambda offers an execution environment for a single function, a small part of an application.

Furthermore, Amazon EC2 offers VMs as a service, but you are responsible for operating them in a secure, scalable and highly available way. Doing so requires you to put a substantial amount of effort into maintenance. By contrast, AWS Lambda offers a fully managed execution environment. AWS manages the underlying infrastructure for you and provides a production-ready infrastructure.

Beyond that, AWS Lambda is billed per execution, and not per second a virtual machine is running. You don't have to pay for unused resources that are waiting for requests or tasks. For example, running a script to check the health of a website every 5 minutes on a virtual machine would cost you a minimum of $4 USD. Executing the same health check with AWS Lambda is free: you don't even exceed the monthly Free Tier of AWS Lambda.

Table 7.1 compares AWS Lambda and virtual machines in detail. You'll find a discussion of AWS Lambda's limitations at the end of the chapter.

Table 7.1 AWS Lambda compared to Amazon EC2

	AWS Lambda	Amazon EC2
Granularity of virtualization	Small piece of code (a function)	A whole operating system
Scalability	Scales automatically. A throttling limit prevents you from creating unwanted charges accidentally and can be increased by AWS support if needed.	As discussed in chapter 17, using an Auto Scaling Group allows you to scale the number of EC2 instances serving requests automatically. But configuring and monitoring the scaling activities is your responsibility.
High availability	Fault tolerant by default. The computing infrastructure spans multiple machines and data centers.	VMs are not highly available by default. Nevertheless, as you will learn in chapter 14 it is possible to set up a highly available infrastructure based on EC2 instances as well.

Table 7.1 AWS Lambda compared to Amazon EC2 *(continued)*

	AWS Lambda	Amazon EC2
Maintenance effort	Almost zero. You need only to configure your function.	You are responsible for maintaining all layers between your virtual machine's operating system and your application's runtime environment.
Deployment effort	Almost zero due to a well-defined API	Rolling out your application to a fleet of VMs is a challenge that requires tools and know-how.
Pricing model	Pay per request as well as execution time and memory	Pay for operating hours of the virtual machines, billed per second.

Looking for limitations and pitfalls of AWS Lambda? Stay tuned: you will find a discussion of Lambda's limitations at the end of the chapter.

That's all you need to know about AWS Lambda to be able to go through the first real-world example. Are you ready?

7.2 Building a website health check with AWS Lambda

Are you responsible for the uptime of a website or application? We do our best to make sure our blog cloudonaut.io is accessible 24/7. An external health check acts as a safety net making sure we, and not our readers, are the first to know when our blog goes down. AWS Lambda is the perfect choice for building a website health check, as you do not need computing resources constantly, but only every few minutes for a few milliseconds. This section guides you through setting up a health check for your website based on AWS Lambda.

In addition to AWS Lambda, we are using the Amazon CloudWatch service for this example. Lambda functions publish metrics to CloudWatch by default. Typically you inspect metrics using charts, and create alarms by defining thresholds. For example, a metric could count failures during the function's execution. On top of that, CloudWatch provides events that can be used to trigger Lambda functions as well. We are using a schedule to publish an event every 5 minutes here.

As shown in figure 7.2, your website health check will consist of three parts:

1. *Lambda function*—Executes a Python script that sends an HTTP request to your website (for example `GET https://cloudonaut.io`) and verifies that the response includes specific text (such as `cloudonaut`).
2. *Scheduled event*—Triggers the Lambda function every 5 minutes. This is comparable to the cron service on Linux.
3. *Alarm*—Monitors the number of failed health checks and notifies you via email whenever your website is unavailable.

You will use the Management Console to create and configure all the necessary parts manually. In our opinion this is a simple way to get familiar with AWS Lambda. You will learn how to deploy a Lambda function in an automated way in section 7.3.

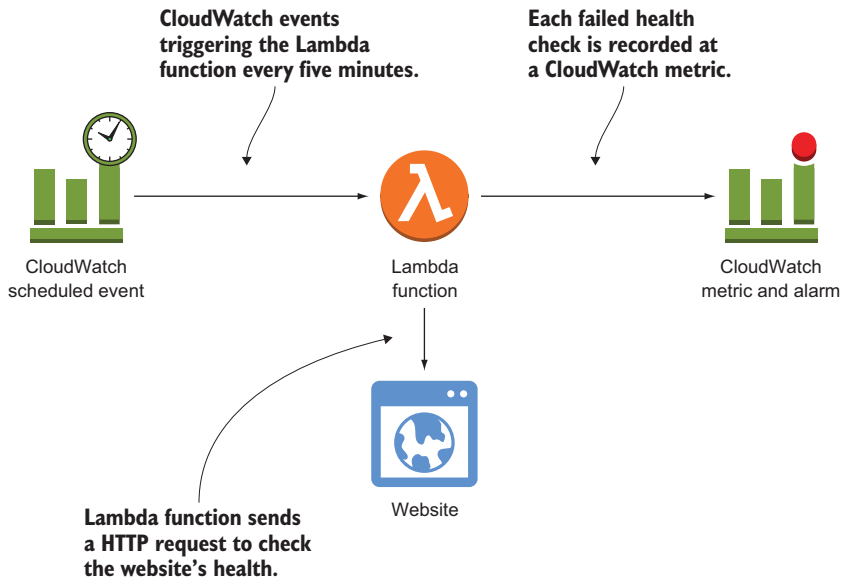


Figure 7.2 The Lambda function performing the website health check is executed every five minutes by a scheduled event. Errors are reported to CloudWatch.

7.2.1 *Creating a Lambda function*

The following step-by-step instructions guide you through setting up a website health check based on AWS Lambda. Open AWS Lambda in the Management Console: https://console.aws.amazon.com/lambda/home. Click Create a function to start the Lambda function wizard, as shown in figure 7.3.

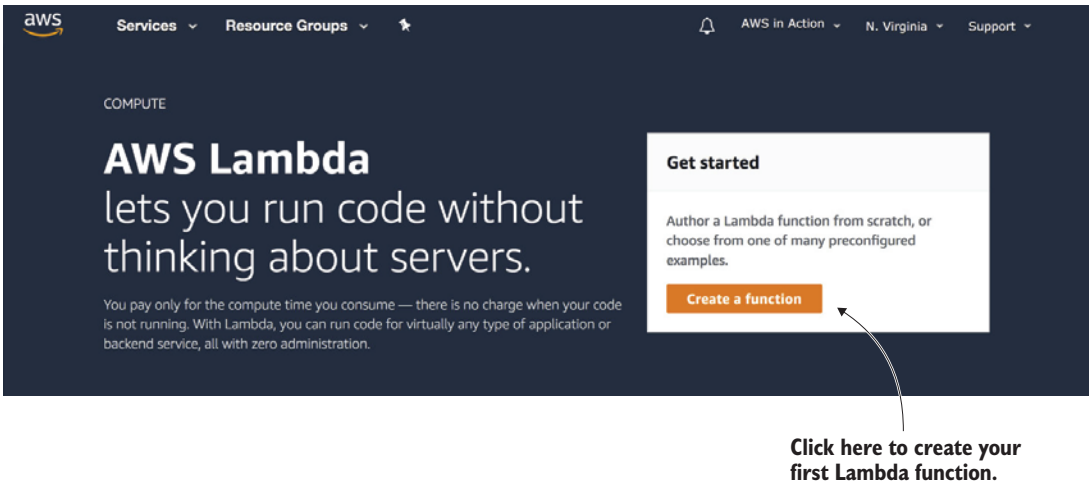


Figure 7.3 Welcome screen: ready to create your first Lambda function

AWS provides blueprints for various use cases, including the code and the Lambda function configuration. We will use one of these blueprints to create a website health check. Select Blueprints and search for `canary`. Next, click the heading of the `lambda-canary-python3` blueprint. Figure 7.4 illustrates the details.

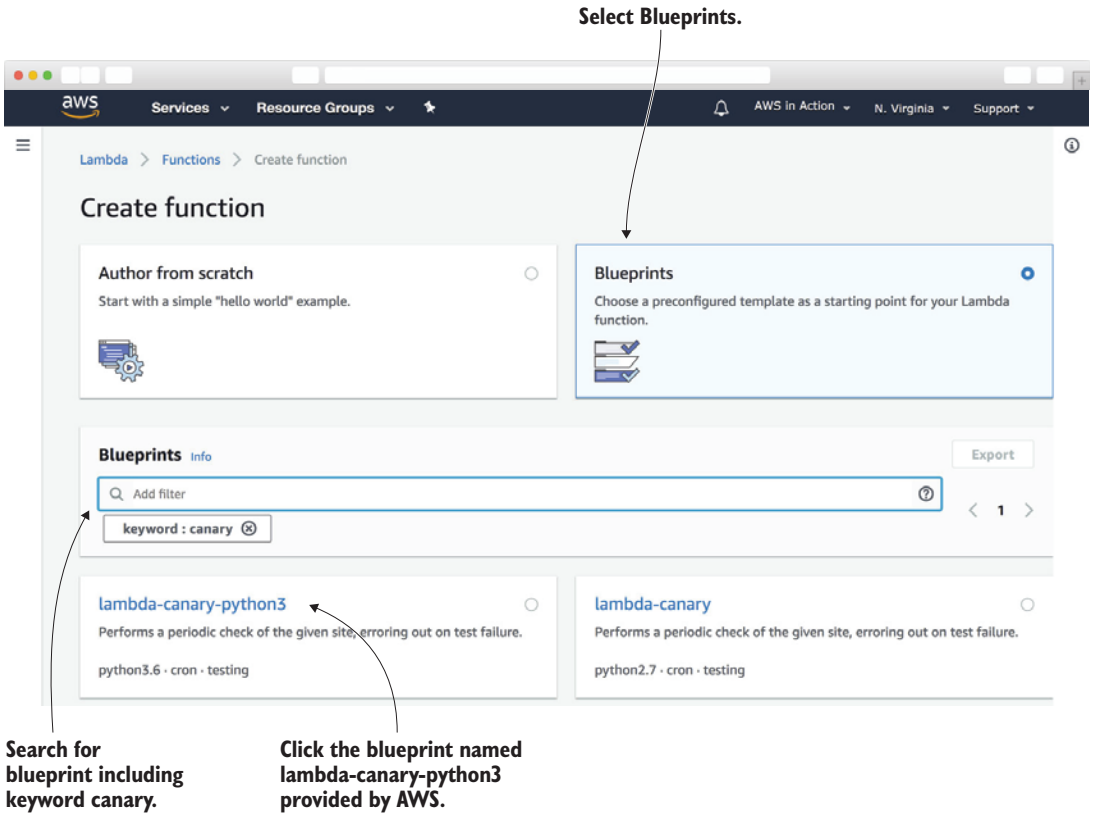


Figure 7.4 Creating a Lambda function based on a blueprint provided by AWS

In the next step of the wizard, you need to specify a name for your Lambda function, as shown in figure 7.5. The function name needs to be unique within your AWS account as well as within the current region US East (N. Virginia), and is limited to 64 characters. To invoke a function via the API you need to provide the function name, for example. Type in `website-health-check` as the name for your Lambda function.

Select Create a custom role as shown in figure 7.5 to create an IAM role for your Lambda function. You will learn how your Lambda function makes use of the IAM role in section 7.3.

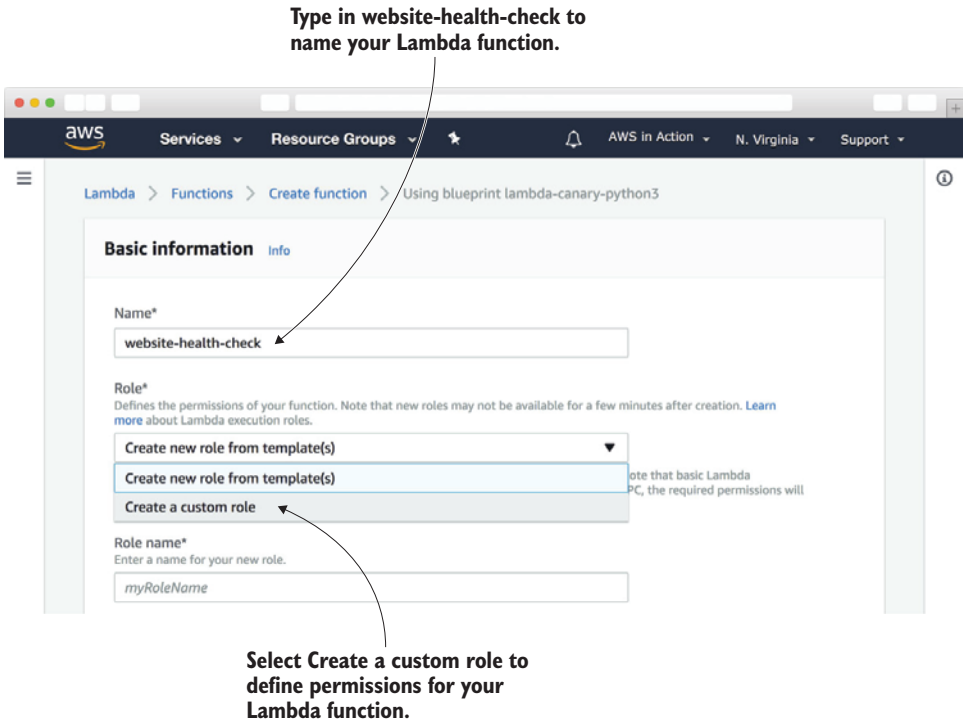


Figure 7.5 Creating a Lambda function: choose a name and define an IAM role

Figure 7.6 illustrates the steps to create a basic IAM role granting your Lambda function write access to CloudWatch logs:

1. Select Create a new IAM role.
2. Keep the role name `lambda_basic_execution`.
3. Click the Allow button.
4. Select the role `lambda_basic_execution` from the drop-down list of existing roles.

You have now specified a name and an IAM role for your Lambda function. Next, you will configure the scheduled event that will trigger your health check repeatedly. We will use an interval of 5 minutes in this example. Figure 7.7 shows the settings you need.

1. Select Create a new rule to create a scheduled event rule.
2. Type in `website-health-check` as the name for the rule.
3. Enter a description that will help you to understand what is going on if you come back later.
4. Select Schedule expression as the rule type. You will learn about the other option, Event pattern, at the end of this chapter.
5. Use `rate(5 minutes)` as the schedule expression.
6. Don't forget to enable the trigger by checking the box at the bottom.

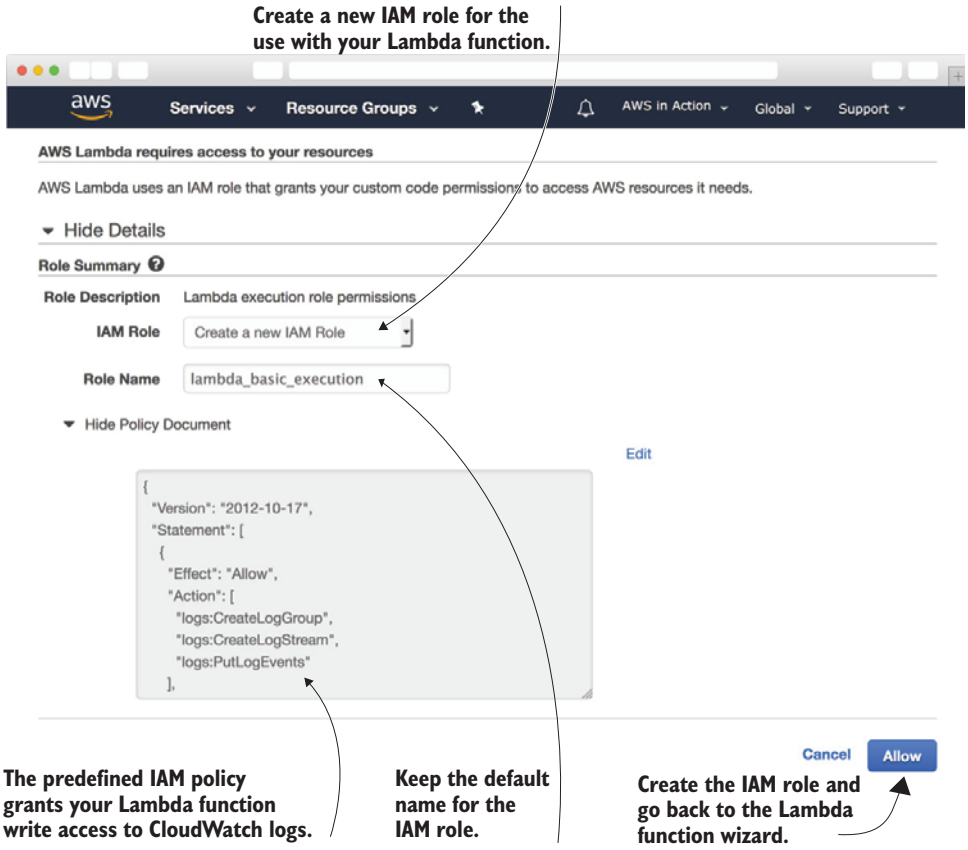


Figure 7.6 Creating an IAM role allowing write access to CloudWatch logs for your Lambda function

You define recurring tasks without the need for a specific time using a *schedule expression* in form of `rate(`*$value $unit*`)`. For example, you could trigger a task every 5 minutes, every hour, or once a day. *$value* needs to be a positive integer value. Use `minute`, `minutes`, `hour`, `hours`, `day`, or `days` as the unit. For example, instead of triggering the website health check every 5 minutes, you could use `rate(1 hour)` as the schedule expression to execute the health check every hour. Note that frequencies of less than one minute are not supported.

It is also possible to use the crontab format when defining a schedule expression.

```
cron($minutes $hours $dayOfMonth $month $dayOfWeek $year)

# Invoke a Lambda function at 08:00am (UTC) everyday
cron(0 8 * * ? *)

# Invoke a Lambda function at 04:00pm (UTC) every monday to friday
cron(0 16 ? * MON-FRI *)
```

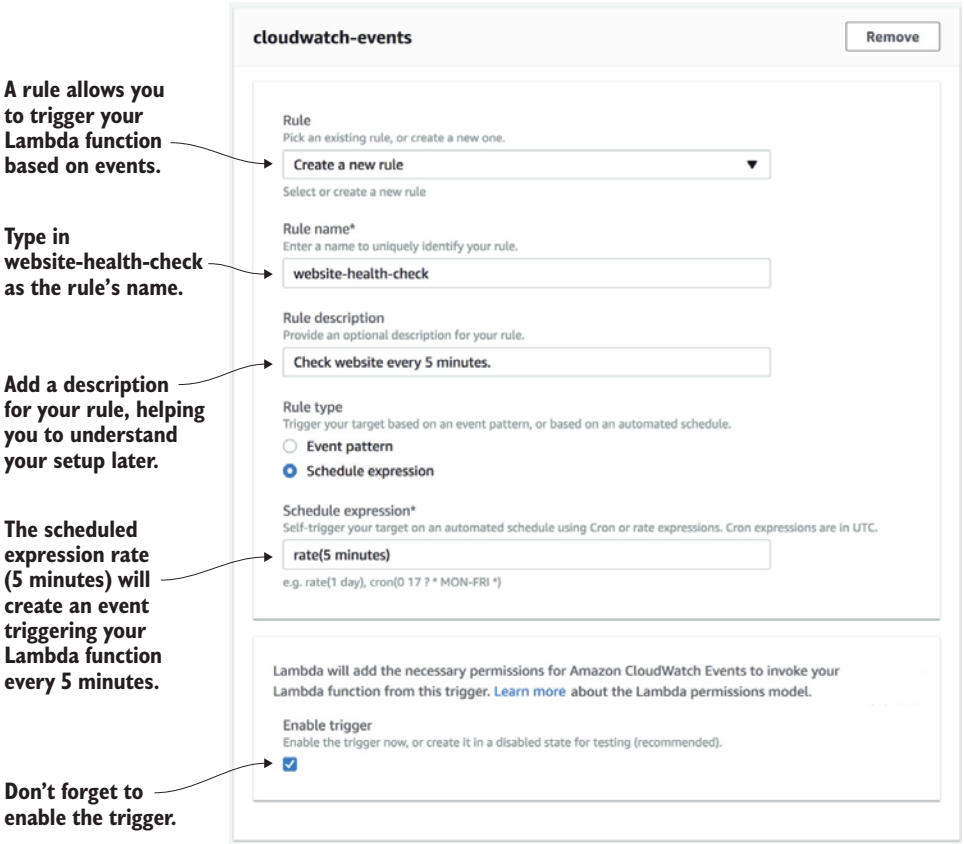


Figure 7.7 Configuring a scheduled event triggering your Lambda function every 5 minutes

See "Schedule Expressions Using Rate or Cron" at http://mng.bz/o35b for more details.

Your Lambda function is missing an integral part: the code. As you are using a blueprint, AWS has inserted the Python code implementing the website health check for you, as shown in figure 7.8.

The Python code references two environment variables: `site` and `expected`. Environment variables are commonly used to dynamically pass settings to your function.

An environment variable consists of a key and a value. Specify the following environment variables for your Lambda function:

1. `site`—Contains the URL of the website you want to monitor. Use https://cloudonaut.io if you do not have a website to monitor yourself.
2. `expected`—Contains a text snippet that must be available on your website. If the function doesn't find this text, the health check fails. Use `cloudonaut` if you are using https://cloudonaut.io as site.

The Lambda function is reading the environment variables during its execution:

```
SITE = os.environment['site']
EXPECTED = os.environment['expected']
```

After defining the environment variables for your Lambda function, click the Create function button at the bottom of the screen.

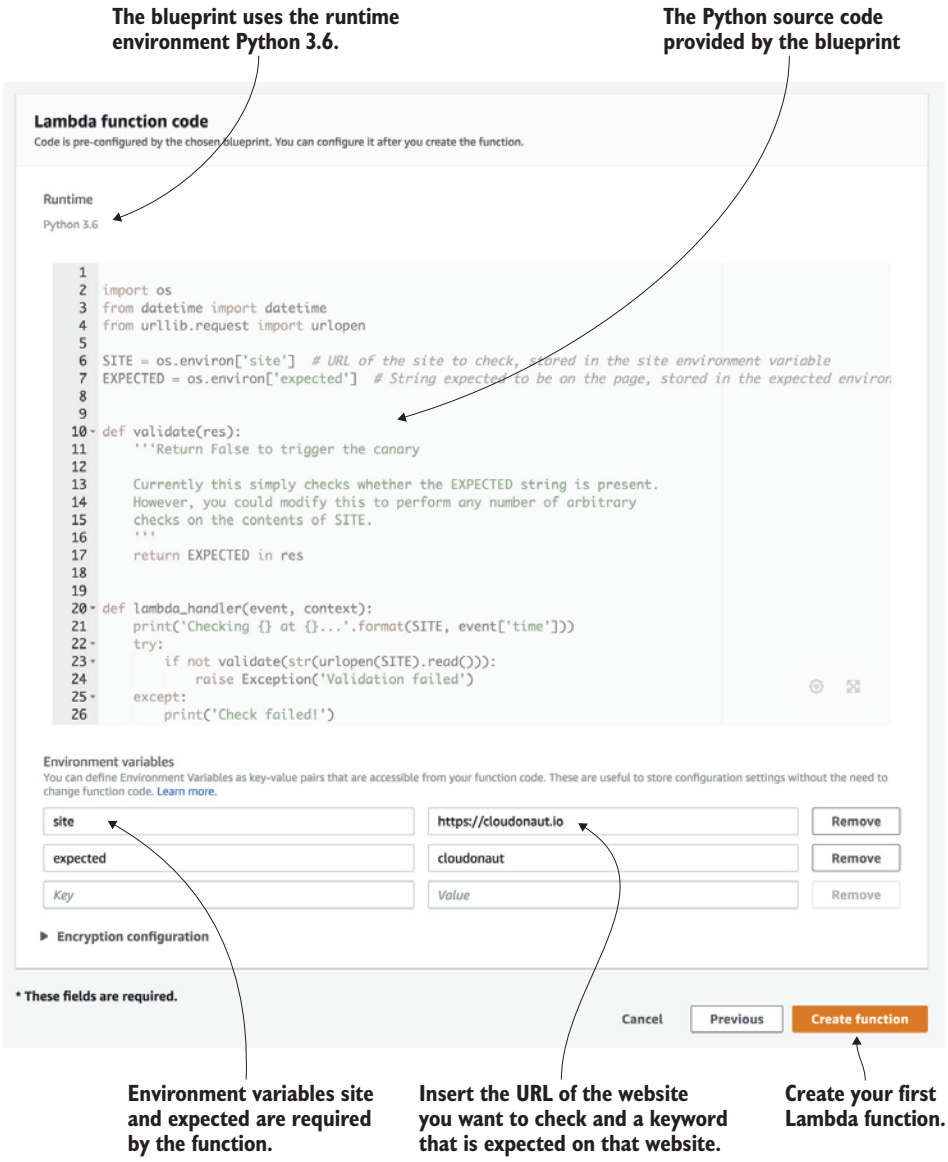


Figure 7.8 The predefined code implementing the website health check and environment variables to pass settings to the Lambda function

Congratulations, you have successfully created a Lambda function. Every 5 minutes, the function is invoked automatically and executes a health check for your website. Next, you will learn how to monitor your Lambda function and get notified via email whenever the health check fails.

7.2.2 Use CloudWatch to search through your Lambda function's logs

How do you know whether your website health check is working correctly? How do you even know if your Lambda function has been executed? It is time to look at how to monitor a Lambda function. You will learn how to access your Lambda function's log messages first. Afterward, you will create an alarm notifying you if your function fails.

Open the Monitoring tab in the details view of your Lambda function. You will find a chart illustrating the number of times your function has been invoked. Use the Reload button of the chart after a few minutes, if the chart isn't showing any invocations. To go to your Lambda function's logs, click View logs in CloudWatch as shown in figure 7.9.

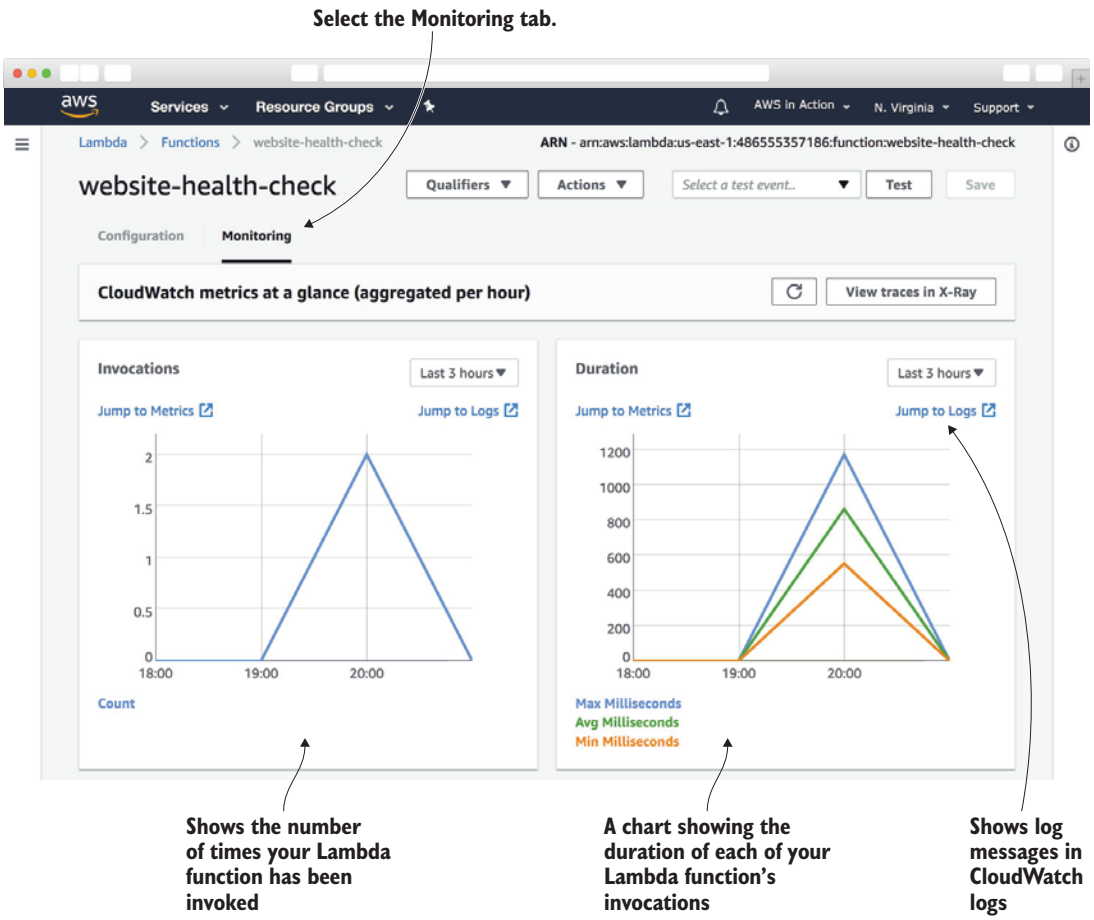


Figure 7.9 Monitoring overview: get insights into your Lambda function's invocations.

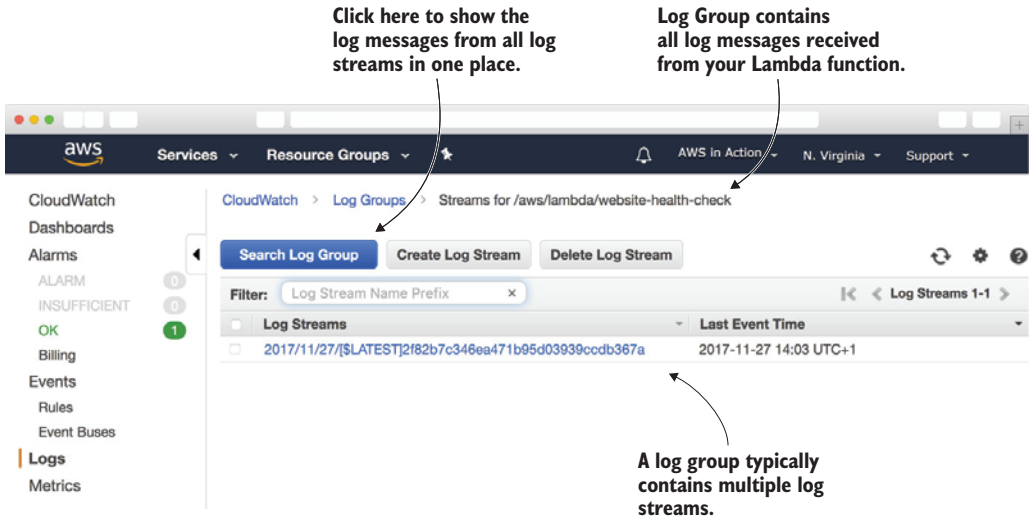


Figure 7.10 A log group collects log messages from a Lambda function stored in multiple log streams.

By default, your Lambda function sends log messages to CloudWatch. Figure 7.10 shows the log group named `/aws/lambda/website-health-check` that was created automatically and collects the logs from your function. Typically, a log group contains multiple log streams, allowing the log group to scale. Click Search Log Group to view the log messages from all streams in one view.

All log messages are presented in the overview of log streams, as shown in figure 7.11. You should be able to find a log message `Check passed!` indicating that the website health check was executed and passed successfully, for example.

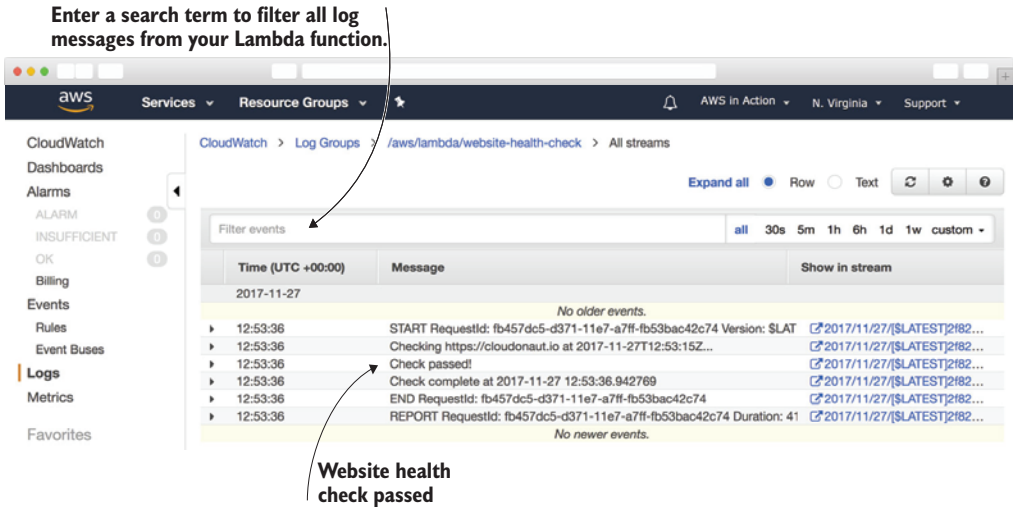


Figure 7.11 CloudWatch shows the log messages of your Lambda function.

The log messages show up after a delay of a few minutes. Reload the table if you are missing any log messages.

Being able to search through log messages in a centralized place is handy when debugging Lambda functions, especially if you are writing your own code. When using Python, you can use `print` statements or use the `logging` module to send log messages to CloudWatch out-of-the-box.

7.2.3 Monitoring a Lambda function with CloudWatch metrics and alarms

The Lambda function checks the health of your website every 5 minutes. A log message with the result of each health check is written to CloudWatch. But how do you get notified via email if the health check fails? Each Lambda function publishes the metrics listed in table 7.2 to CloudWatch by default:

Table 7.2 The CloudWatch metrics published by each Lambda function

Name	Description
Invocations	Counts the number of times a function is invoked. Includes successful and failed invocations.
Errors	Counts the number of times a function failed due to errors inside the function. For example, exceptions or timeouts.
Duration	Measures how long the code takes to run, from the time when the code starts executing to when it stops executing.
Throttles	As discussed at the beginning of the chapter, there is a limit for how many copies of your Lambda function are running at one time. This metric counts how many invocations have been throttled due to reaching this limit. Contact AWS support to increase the limit if needed.

Whenever the website health check fails, the Lambda function returns an error, increasing the count of the Errors metric. You will create an alarm notifying you via email whenever this metric counts more than 0 errors. In general, we recommend to create an alarm on the following metrics to monitor your Lambda functions: Errors and Throttles.

The following steps guide you through creating a CloudWatch alarm to monitor your website health checks. Your Management Console still shows the CloudWatch service. Select Alarms from the sub-navigation menu. Did you create a billing alarm in chapter 1? If so, the alarm will be listed here. Next, click Create Alarm as shown in figure 7.12.

First of all, you need to select the Errors metric for your Lambda function. Click By Function Name under Lambda. Figure 7.13 shows an overview.

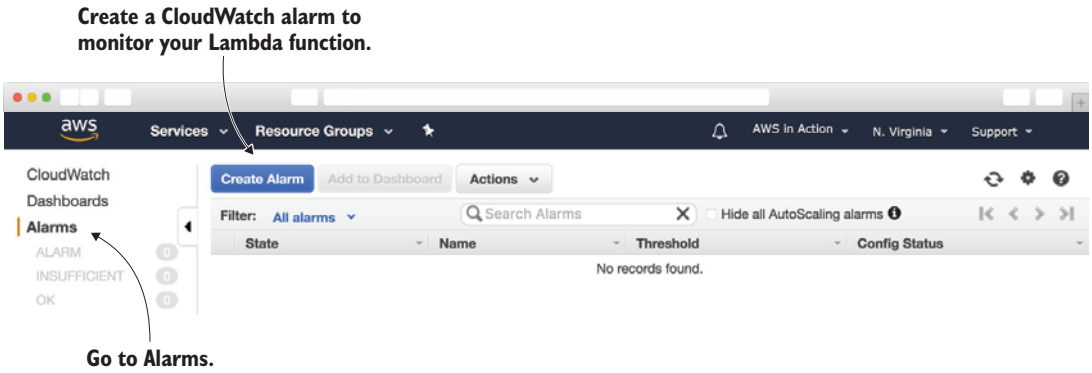


Figure 7.12 Starting the wizard to create a CloudWatch alarm to monitor a Lambda function

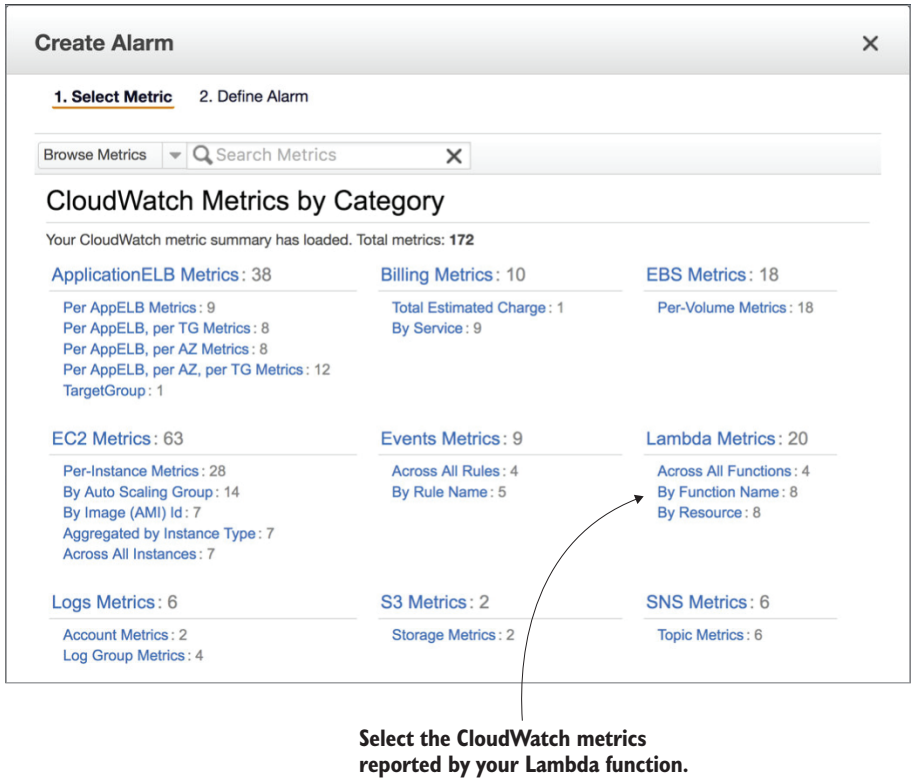


Figure 7.13 Selecting the Lambda metric to create an alarm

The next step is to select and prepare the metric for the alarm. Figure 7.14 shows the necessary steps:

1. Search for metrics belonging to your Lambda function `website-health-check`.
2. Select the Errors metric.
3. For the statistics, use Sum. This will set the alarm to trigger based on the total number of errors.
4. Choose 5 minutes for how often the alarm is updated (the time bucket).
5. Select a timespan from 5 minutes ago to now.
6. Click the Next button to proceed with the next step.

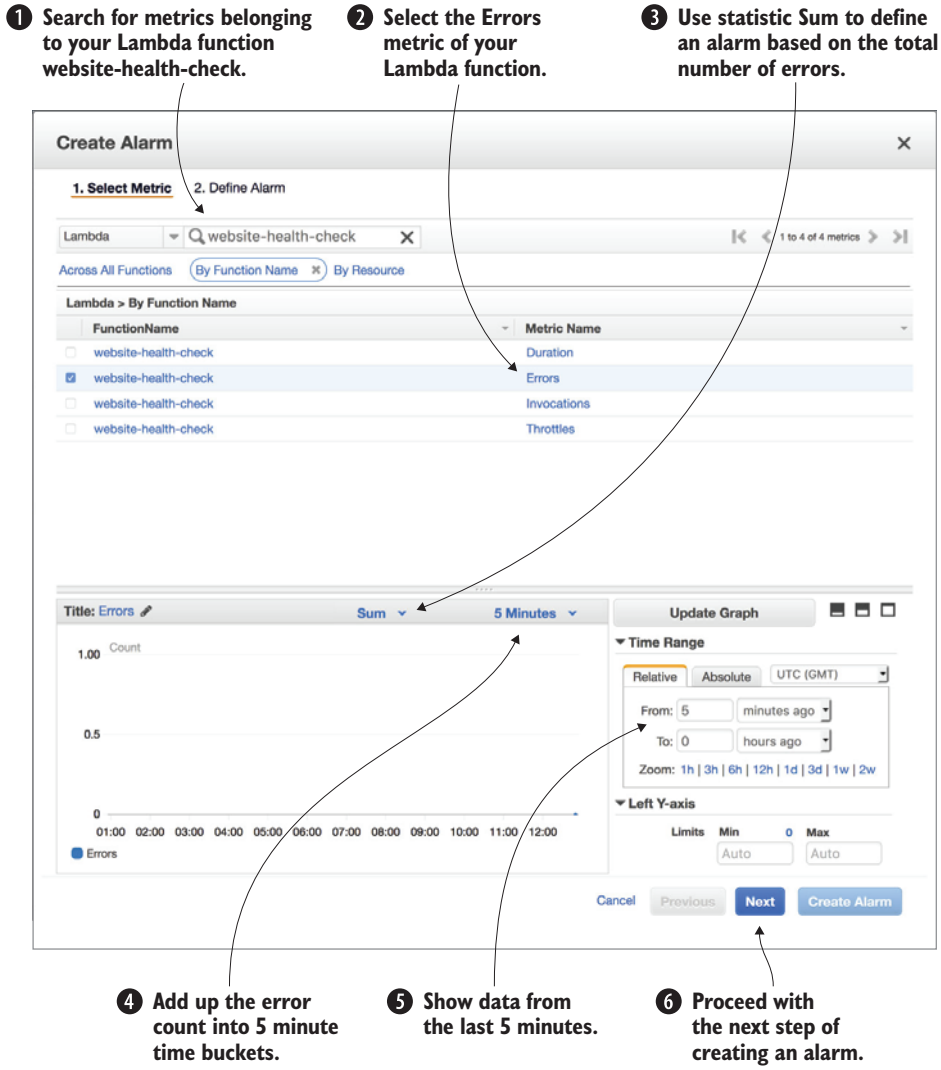


Figure 7.14 Selecting and preparing the metric view for the alarm

To create the alarm you need to define a name, a threshold, and the actions to be performed. Figure 7.15 shows the details.

1. Type in `website-health-check-error` as Name for the alarm.
2. Define a Description for the alarm.
3. Specify a threshold for the alarm. Use the drop-down boxes to set up the alarm like this: `Whenever Errors is > 0 for 1 out of 1 datapoints.`
4. Keep the default State is ALARM for the action.
5. Create a new notification list by clicking New List and typing in `website-health-check` as the name for the list.
6. Enter your email into the Email list.
7. Click the Create Alarm button.

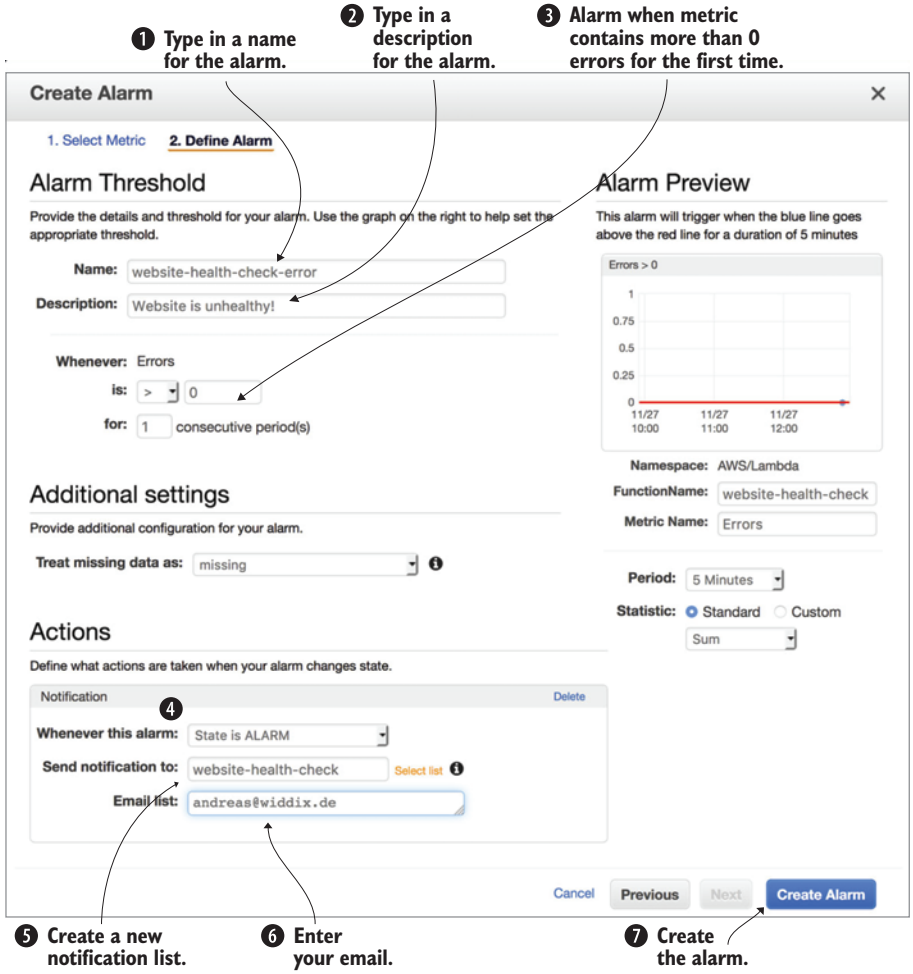


Figure 7.15 Creating an alarm by defining a threshold and defining an alarm action to send notifications via email

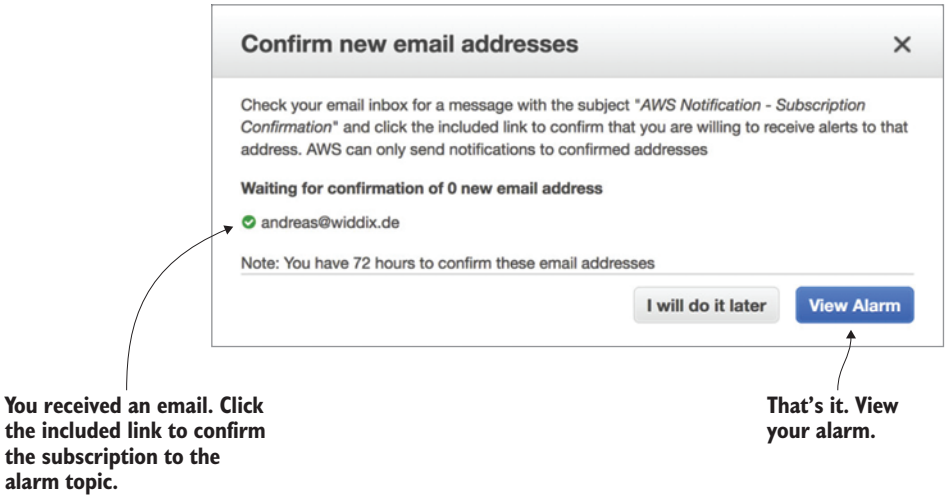


Figure 7.16 Check your inbox and confirm your subscription to the notification list.

As shown in figure 7.16, you will be sent an email including a confirmation link. Check your inbox and click the link to confirm your subscription to the notification list. Afterward, click the View Alarm button.

You will receive an alarm via email whenever your website health check fails. As you probably do not want to take down your website just so you can check that the alarm, you could change the function's environment variables. For example, you could change the `expected` value to a text snippet that your website does not include. A few minutes after changing the environment variable, you will receive an alarm via email.

Cleaning up

Open your Management Console and follow these steps to delete all the resources you have created during this section.

1. Go to the AWS Lambda service and delete the function named `website-health-check`.
2. Open the AWS CloudWatch service, select Logs from the sub-navigation menu, and delete the log group `/aws/lambda/website-health-check`.
3. Go to the AWS CloudWatch service, select Events from the sub-navigation menu, and delete the rule `website-health-check`.
4. Select Alarms from the sub navigation, and delete the alarm `website-health-check-error`.
5. Jump to the AWS IAM service, select Roles from the sub-navigation menu, and delete the role `lambda_basic_execution`.

7.2.4 Accessing endpoints within a VPC

As illustrated in figure 7.17, Lambda functions run outside your networks defined with VPC by default. However, Lambda functions are connected to the internet and therefore able to access other services. That's exactly what you have been doing when creating a website health check: the Lambda function was sending HTTP requests over the internet.

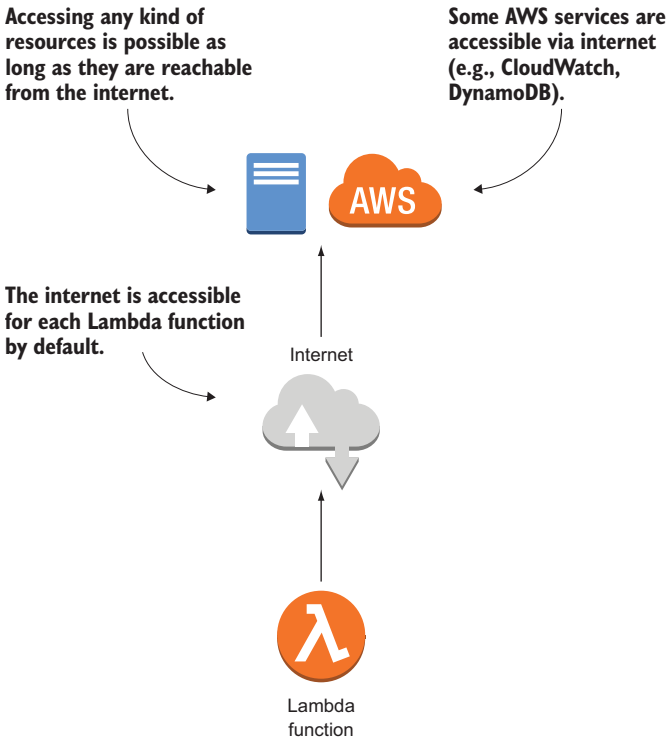


Figure 7.17 By default a Lambda function is connected to the internet and running outside your VPCs.

So what do you do when you have to reach a resource running in a private network within your VPC? For example, if you want to run a health check for an internal website? If you add network interfaces to your Lambda function, the function can access resources within your VPCs as shown in figure 7.18.

To do so you have to define the VPC, the subnets, as well as security groups for your Lambda function. See "Configuring a Lambda Function to Access Resources in an Amazon VPC" at http://docs.aws.amazon.com/lambda/latest/dg/vpc.html for more details. We have been using the ability to access resources within a VPC to access databases in various projects.

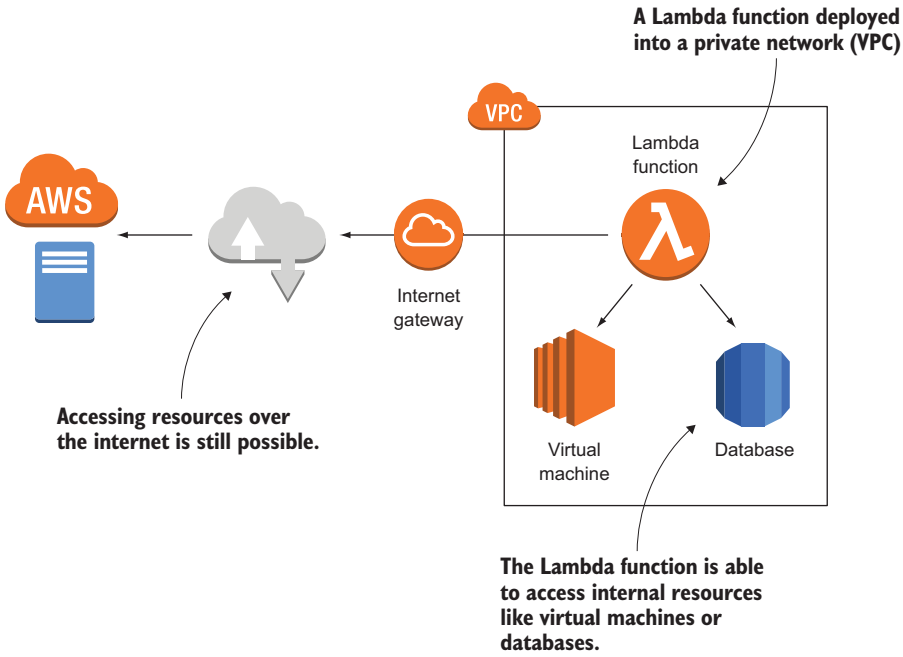


Figure 7.18 Deploying a Lambda function into your VPC allows you to access internal resources (such as database, virtual machines, and so on).

AWS recommends placing a Lambda function in a VPC only when absolutely necessary to reach a resource that is not accessible otherwise. Placing a Lambda function in a VPC increases complexity, especially when scaling to a large number of concurrent executions. For example, the number of available private IP addresses in a VPC is limited, but a Lambda function will need multiple private IP addresses to be able to scale the number of concurrent invocations.

7.3 Adding a tag containing the owner of an EC2 instance automatically

After using one of AWS's predefined blueprints to create a Lambda function, you will implement a Lambda function from scratch in this section. We are strongly focused on setting up your cloud infrastructure in an automated way. That's why you will learn how to deploy a Lambda function and all its dependencies without needing the Management Console.

Are you working in an AWS account together with your colleagues? Have you ever wondered who launched a certain EC2 instance? Sometimes you need to find out the owner of an EC2 instance for the following reasons:

- Double-checking if it is safe to terminate an unused instance without losing relevant data.
- Reviewing changes to an instance's configuration with its owner (such as making changes to the firewall configuration).
- Attributing costs to individuals, projects, or departments.
- Restricting access to an instance (for example, so only the owner is allowed to terminate an instance).

Adding a tag that states who owns an instance solves all these use cases. A tag can be added to an EC2 instance or almost any other AWS resource, and consists of a key and a value. Tags can be used to add information to a resource, filter resources, attribute costs to resources, as well as to restrict access. See "Tagging Your Amazon EC2 Resources" at http://mng.bz/pEJN for more details.

It is possible to add tags specifying the owner of an EC2 instance manually. But sooner or later, someone will forget to add the owner tag. There is a better solution for that! You will implement and deploy a Lambda function that adds a tag containing the name of the user who launched an EC2 instance automatically in the following section. But how do you execute a Lambda function every time an EC2 instance is launched, so that you can add the tag?

7.3.1 Event-driven: Subscribing to CloudWatch events

CloudWatch consists of multiple parts. You have already learned about metrics, alarms, and logs during this chapter. Another feature built into CloudWatch is called *events*. Whenever something changes in your infrastructure, an event is generated in near real-time:

- CloudTrail emits an event for every call to the AWS API.
- EC2 emits events whenever the state of an EC2 instances changes (such as when the state changes from pending to running).
- AWS emits an event to notify you of service degradations or downtimes.

Whenever you launch a new EC2 instance, you are sending a call to the AWS API. Subsequently, CloudTrail generates a CloudWatch event. Our goal is to add a tag to every new EC2 instance. Therefore, we are executing a function for every event that indicates the launch of a new EC2 instance. To trigger a Lambda function whenever such an event occurs, you need a rule. As illustrated in figure 7.19, the rule matches incoming events and routes them to a target, a Lambda function in our case.

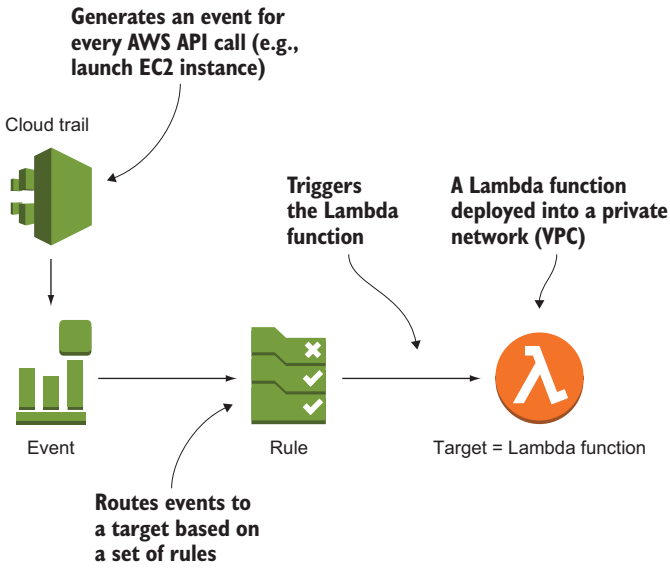


Figure 7.19 CloudTrail generates an event for every AWS API call, a rule routes the event to the Lambda function

Listing 7.1 shows some of the event details generated by CloudTrail whenever someone launches an EC2 instance. For our case, we're interested in the following information:

- `detail-type`—The event has been created by CloudTrail.
- `source`—The EC2 service is the source of the event.
- `eventName`—The event name `RunInstances` indicates that the event was generated because of an AWS API call launching an EC2 instance.
- `userIdentity`—Who called the AWS API to launch an instance?
- `responseElements`—The response from the AWS API when launching an instance. This includes the ID of the launched EC2 instance that we will need to add a tag to the instance later.

Listing 7.1 CloudWatch event generated by CloudTrail when launching an EC2 instance

```
{
  "version": "0",
  "id": "8a50bfef-33fd-2ea3-1056-02ad1eac7210",
  "detail-type": "AWS API Call via CloudTrail",
  "source": "aws.ec2",
  "account": "XXXXXXXXXXX",
  "time": "2017-11-30T09:51:25Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "eventVersion": "1.05",
    "userIdentity": {
```

CloudTrail generated the event

Someone sent a call to the AWS API affecting the EC2 service.

Information about the user who launched the EC2 instance

```
      "type": "IAMUser",
      "principalId": "...",
      "arn": "arn:aws:iam::XXXXXXXXXXXX:user/myuser",
      "accountId": "XXXXXXXXXXXX",
      "accessKeyId": "...",
      "userName": "myuser",
      "sessionContext": {
        "attributes": {
          "mfaAuthenticated": "false",
          "creationDate": "2017-11-30T09:51:05Z"
        }
      },
      "invokedBy": "signin.amazonaws.com"
    },
    "eventTime": "2017-11-30T09:51:25Z",
    "eventSource": "ec2.amazonaws.com",
    "eventName": "RunInstances",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "XXX.XXX.XXX.XXX",
    "userAgent": "signin.amazonaws.com",
    "requestParameters": {
      [...]
    },
    "responseElements": {
      "requestId": "327f5231-c65a-468c-83a8-b00b7c949f78",
      "reservationId": "r-0234df5d03e3ad6a5",
      "ownerId": "XXXXXXXXXXXX",
      "groupSet": {},
      "instancesSet": {
        "items": [
          {
            "instanceId": "i-06133867ab0f704e7",
            "imageId": "ami-55ef662f",
            [...]
          }
        ]
      }
    },
    "requestID": "327f5231-c65a-468c-83a8-b00b7c949f78",
    "eventID": "134d35c6-6a76-49b9-9ff8-5a4130474b6f",
    "eventType": "AwsApiCall"
  }
}
```

ID of the user who launched the EC2 instance

Event was generated because a RunInstances call (used to launch an EC2 instance) was processed by the AWS API.

Response of the AWS API when launching the instance

ID of the launched EC2 instance

A rule consists of an event pattern for selecting events, along with a definition of one or multiple targets. The following pattern selects all events from CloudTrail generated by an AWS API call affecting the EC2 service. The pattern matches three attributes from the event described in listing 7.1: `detail-type`, `source`, and `eventName`.

Listing 7.2 Rule to filter events from CloudTrail

```
{
  "detail-type": [
    "AWS API Call via CloudTrail"
```

Filter events from CloudTrail caused by AWS API calls.

```
  ],
  "source": [
    "aws.ec2"
  ],
  "detail": {
    "eventName": [
      "RunInstances"
    ]
  }
}
```

Filter events from the EC2 service.

Filter events with event name RunInstances, which is the AWS API call to launch an EC2 instance.

Defining filters on other event attributes is possible as well, in case you are planning to write another rule in the future. The rule format stays the same.

When specifying an event pattern, we typically use the following fields, which are included in every event:

- `source`—The namespace of the service which generated the event. See "Amazon Resource Names (ARNs) and AWS Service Namespaces" at http://mng.bz/GFGm for details.
- `detail-type`—Categorizes the event in more detail.

See "Event Patterns in CloudWatch Events" at http://mng.bz/37Ux for more detailed information.

You have now defined the events that will trigger your Lambda function. Next, you will implement the Lambda function.

7.3.2 Implementing the Lambda function in Python

Implementing the Lambda function to tag an EC2 instance with the owner's user name is simple. You will need to write no more than 10 lines of Python code. The programming model for a Lambda function depends on the programming language you choose. Although we are using Python in our example, you will be able to apply what you've learned when implementing a Lambda function in Java, Node.js, C#, or Go. As shown in the next listing, your function written in Python needs to implement a well-defined structure.

Listing 7.3 Lambda function written in Python

The name of the Python function, which is referenced by the AWS Lambda as the function handler. The event parameter is used to pass the CloudWatch event and the context parameter includes runtime information.

It is your job to implement the function.

```
def lambda_handler(event, context):
  # Insert your code
  return
```

Use return to end the function execution. It is not useful to hand over a value in this scenario, as the Lambda function is invoked asynchronously by a CloudWatch event.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. Switch to the `chapter07` directory, which includes all files needed for this example.

Time to write some Python code! The following listing for `lambda_function.py` shows the function, which receives an event from CloudTrail indicating that an EC2 has been launched recently, and adds a tag including the name of the instance's owner. The AWS SDK for Python 3.6, named `boto3`, is provided out-of-the-box in the Lambda runtime environment for Python 3.6. In this example you are using the AWS SDK to create a tag for the EC2 instance `ec2.create_tags(...)`. See the Boto 3 Documentation at https://boto3.readthedocs.io/en/latest/index.html if you are interested in the details of `boto3`.

Listing 7.4 Lambda function adding a tag to EC2 instance

Creates an AWS SDK client to manage the EC2 service

The name of the function used as entry point for the Lambda function

Extracts the user's name from the CloudTrail event

Extracts the instance's ID from the CloudTrail event

Adds a tag to the EC2 instance using the key owner and the user's name as value

```
import boto3
ec2 = boto3.client('ec2')

def lambda_handler(event, context):

  userName = event['detail']['userIdentity']['arn'].split('/')[1]
  instanceId = event['detail']['responseElements']
➥ ['instancesSet']['items'][0]['instanceId']
  print("Adding owner tag " + userName + " to instance " + instanceId + ".")
  ec2.create_tags(Resources=[instanceId,],
➥ Tags=[{'Key': 'Owner', 'Value': userName},])
  return
```

After implementing your function in Python, the next step is to deploy the Lambda function with all its dependencies.

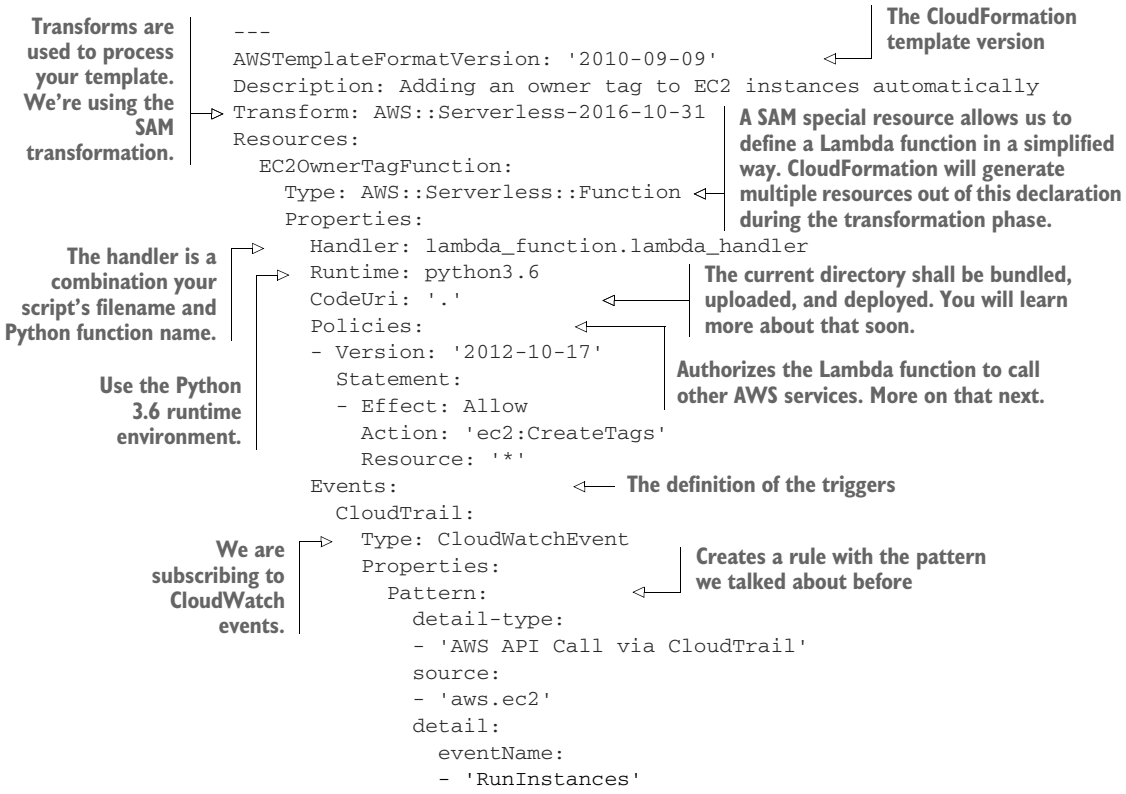
7.3.3 Setting up a Lambda function with the Serverless Application Model (SAM)

You have probably noticed that we are huge fans of automating infrastructures with CloudFormation. Using the Management Console is a perfect way to take the first step when learning about a new service on AWS. But leveling up from manually clicking through a web interface to fully automating the deployment of your infrastructure should be your second step.

AWS released the *Serverless Application Model* (SAM) in 2016. SAM provides a framework for serverless applications, extending plain CloudFormation templates to make it easier to deploy Lambda functions.

This listing shows how to define a Lambda function using SAM and a CloudFormation template.

Listing 7.5 Defining a Lambda function with SAM within a CloudFormation template



7.3.4 Authorizing a Lambda function to use other AWS services with an IAM role

Lambda functions typically interact with other AWS services. For instance, they might write log messages to CloudWatch allowing you to monitor and debug your Lambda function. Or they might create a tag for an EC2 instance, as in the current example. Therefore, calls to the AWS APIs need to be authenticated and authorized. Figure 7.20 shows a Lambda function assuming an IAM role to be able to send authenticated and authorized requests to other AWS services.

Temporary credentials are generated based on the IAM role and injected into each invocation via environment variables (such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`). Those environment variables are used by the AWS SDK to sign requests automatically.

You should follow the least-privilege principle: your function should only be allowed to access services and actions that are needed to perform the function's task.

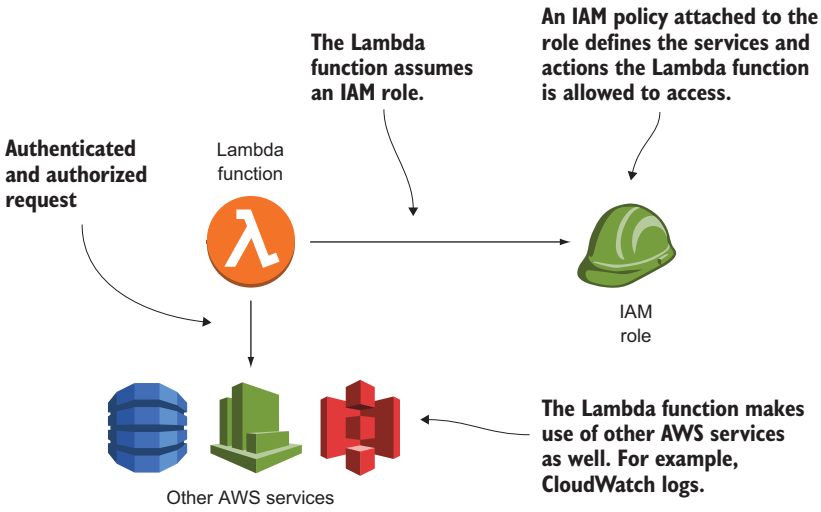


Figure 7.20 A Lambda function assumes an IAM role to authenticate and authorize requests to other AWS services.

You should specify a detailed IAM policy granting access to specific actions and resources.

Listing 7.6 shows an excerpt from the Lambda function's CloudFormation template based on SAM. When using SAM, an IAM role is created for each Lambda function by default. A managed policy that grants write access to CloudWatch logs is attached to the IAM role by default as well. Doing so allows the Lambda function to write to CloudWatch logs.

So far the Lambda function is not allowed to create a tag for the EC2 instance. You need a custom policy granting access to the `ec2:CreateTags`.

Listing 7.6 A custom policy for adding tags to EC2 instances

```
# [...]
EC2OwnerTagFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: lambda_function.lambda_handler
    Runtime: python3.6
    CodeUri: '.'
    Policies:
    - Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Action: 'ec2:CreateTags'
        Resource: '*'
    # [...]
```

Lets you define your own custom IAM policy that will be attached to the Lambda function's IAM role

The statement allows ...

...creating tags

...for all resources.

In case you implement another Lambda function in the future, make sure you create an IAM role granting access to all the services your function needs to access (such as reading objects from S3, writing data to a DynamoDB database, and so on). Revisit section 6.3 if you want to recap the details of IAM.

7.3.5 Deploying a Lambda function with SAM

To deploy a Lambda function, you need to upload the deployment package to S3. The deployment package is a zip file including your code as well as additional modules. Afterward, you need to create and configure the Lambda function as well as all the dependencies (the IAM role, event rule, and so on). Using SAM in combination with the AWS CLI allows you to accomplish both tasks.

First of all you, need to create an S3 bucket to store your deployment packages. Use the following command, replacing *$yourname* with your name to avoid name conflicts with other readers.

```
$ aws s3 mb s3://ec2-owner-tag-$yourname
```

The next step is to create the deployment package and upload the package to S3. Execute the following command in your terminal to do so. A copy of your template is stored as `output.yaml`, with a reference to the deployment package uploaded to S3.

```
$ aws cloudformation package --template-file template.yaml \
➥ --s3-bucket ec2-owner-tag-$yourname --output-template-file output.yaml
```

By typing in the following command in your terminal, you are deploying the Lambda function. This results in a CloudFormation stack named `ec2-owner-tag`. Make sure your working directory is the code directory `chapter07` containing the `template.yaml` and `lambda_function.py` files.

```
$ aws cloudformation deploy --stack-name ec2-owner-tag \
➥ --template-file output.yaml --capabilities CAPABILITY_IAM
```

You are a genius! Your Lambda function is up and running. Launch an EC2 instance and you will find a tag with your user name `myuser` attached after a few minutes.

Cleaning up

If you have launched an EC2 instance to test your Lambda function, don't forget to terminate the instance afterward.

Otherwise it is quite simple to delete the Lambda function and all its dependencies. Just execute the following command in your terminal. Replace *$yourname* with your name.

```
$ aws cloudformation delete-stack --stack-name ec2-owner-tag
$ aws s3 rb s3://ec2-owner-tag-$yourname --force
```

7.4 What else can you do with AWS Lambda?

In the last part of the chapter, we would like to share what else is possible with AWS Lambda, starting with Lambda's limitations and insights into the serverless pricing model. We will end with three use cases for serverless applications we have built for our consulting clients.

7.4.1 What are the limitations of AWS Lambda?

Each invocation of your Lambda function needs to complete within a maximum of 300 seconds. This means the problem you are solving with your function needs to be small enough to fit into the 300-second limit. It is probably not possible to download 10 GB of data from S3, process the data, and insert parts of the data into a database within a single invocation of a Lambda function. But even if your use case fits into the 300-second constraint, make sure that it does under all circumstances. Here's a short anecdote from one of our first serverless projects: We built a serverless application that pre-processed analytics data from news sites. The Lambda functions typically processed the data within less than 180 seconds. But when the 2017 U.S. elections came, the volume of the analytics data exploded in a way no one expected. Our Lambda functions were no longer able to complete within 300 seconds. A show stopper for our serverless approach.

AWS Lambda provisions and manages the resources needed to run your function. A new execution context is created in the background every time you deploy a new version of your code, go a long time without any invocations, or when the number of concurrent invocations increases. Starting a new execution context requires AWS Lambda to download your code, initialize a runtime environment, and load your code. This process is called a *cold-start*. Depending on the size of your deployment package, the runtime environment, and your configuration, a cold-start could take from a few milliseconds to a few seconds. Therefore, applications with very strict requirements concerning response times are not good candidates for AWS Lambda. Conversely, there are a lot of use cases where the additional latency caused by a cold-start is acceptable. For example, the two examples in this chapter are not affected by a cold-start at all. To minimize cold-start times, you should keep the size of your deployment package as small as possible, provision additional memory, and use a runtime environment like Python, Node.js, or Go instead of C# or Java.

Another limitation is the maximum amount of memory you can provision for a Lambda function: 3008 MB. If your Lambda function uses more memory, its execution will be terminated.

It is also important to know that CPU and networking capacity are allocated to a Lambda function based on the provisioned memory as well. So if you are running computing or network-intensive work within a Lambda function, increasing the provisioned memory will probably improve performance.

At the same time, the default limit for the maximum size of the compressed deployment package (zip file) is 50 MB. When executing your Lambda function, you can use up to 500 MB non-persistent disk space mounted to `/tmp`.

Look at "AWS Lambda Limits" at http://docs.aws.amazon.com/lambda/latest/dg/limits.html if you want to learn more about Lambda's limitations.

7.4.2 Impacts of the serverless pricing model

When launching a virtual machine, you have to pay AWS for every operating hour, billed in second-intervals. You are paying for the machines no matter if you are using the resource they provide. Even when nobody is accessing your website or using your application, you are paying for the virtual machine.

That's totally different with AWS Lambda. Lambda is billed per request. Costs occur only when someone accesses your website or uses your application. That's a game changer, especially for applications with uneven access patterns, or for applications that are used rarely. Table 7.3 explains the Lambda pricing model in detail.

Table 7.3 AWS Lambda pricing model

	Free Tier	Occurring costs
Number of Lambda function invocations	First 1 million requests every month	$0.0000002 USD per request
Duration billed in 100 ms increments based on the amount of memory you provisioned for your Lambda function	Using the equivalent of 400,000 seconds of a Lambda function with 1 GB provisioned memory every month	$0.00001667 USD for using 1 GB for one second

Free Tier for AWS Lambda

The Free Tier for AWS Lambda does not expire after 12 months. That's a huge difference compared to the Free Tier of other AWS services (such as EC2) where you are only eligible for the Free Tier within the first 12 month after creating an AWS account.

Sounds complicated? Figure 7.21 shows an excerpt of an AWS bill. The bill is from November 2017, and belongs to an AWS account we are using to run a chatbot (see marbot.io). Our chatbot implementation is 100% serverless. The Lambda functions were executed 1.2 million times in November 2017, which results in a charge of $0.04 USD. All our Lambda functions are configured to provision 1536 MB memory. In total, all our Lambda functions have been running for 216,000 seconds, roundabout 60 hours in November 2017. That's still within the Free Tier of 400,000 seconds with 1 GB provisioned memory every month. So in total we had to pay $0.04 for using AWS Lambda in November 2017, which allowed us to serve around 400 customers with our chatbot.

▾ Lambda		$0.04
▾ US East (Northern Virginia) Region		$0.04
AWS Lambda Lambda-GB-Second		$0.00
AWS Lambda - Compute Free Tier - 400,000 GB-Seconds - US East (Northern Virginia)	331,906.500 seconds	$0.00
AWS Lambda Request		$0.04
0.0000000367 USD per AWS Lambda - Requests Free Tier - 1,000,000 Requests - US East (Northern Virginia) (blended price)*	1,209,096 Requests	$0.04

Figure 7.21 Excerpt from our AWS bill from November 2017 showing costs for AWS Lambda

This is only a small piece of our AWS bill, as other services used together with AWS Lambda, for example to store data, add more significant costs to our bill.

Don't forget to compare costs between AWS Lambda and EC2. Especially in a high-load scenario with more than 10 million requests per day, using AWS Lambda will probably result in higher costs compared to using EC2. But comparing infrastructure costs is only one part of what you should be looking at. Consider the total cost of ownership (TOC), including costs for managing your virtual machines, performing load and resilience tests, and automating deployments as well.

Our experience has shown that the total cost of ownership is typically lower when running an application on AWS Lambda compared to Amazon EC2.

The last part of the chapter focuses on additional use cases for AWS Lambda besides automating operational tasks, as you have done thus far.

7.4.3 Use case: Web application

A common use case for AWS Lambda is building a back end for a web or mobile application. As illustrated in figure 7.22, an architecture for a serverless web application typically consists of the following building blocks:

- *Amazon API Gateway*—Offers a scalable and secure REST API that accepts HTTPS requests from your web application's front end or mobile application.
- *AWS Lambda*—Lambda functions are triggered by the API gateway. Your Lambda function receives data from the request and returns the data for the response.
- *Object store and NoSQL database*—For storing and querying data, your Lambda functions typically use additional services offering object storage or NoSQL databases, for example.

Do you want to get started building web applications based on AWS Lambda? We recommend *AWS Lambda in Action* from Danilo Poccia, (Manning, 2016).

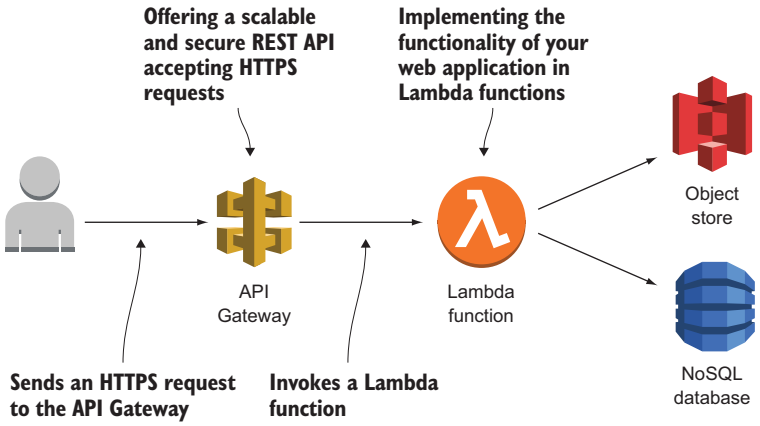


Figure 7.22 A web application build with API Gateway and Lambda

7.4.4 Use case: Data processing

Another popular use case for AWS Lambda is event-driven data processing. Whenever new data is available, an event is generated. The event triggers the data processing needed to extract or transform the data. Figure 7.23 shows an example.

1. The load balancer collects access logs and uploads them to an object store periodically.
2. Whenever an object is created or modified, the object store triggers a Lambda function automatically.
3. The Lambda function downloads the file including the access logs from the object store, and sends the data to an Elasticsearch database to be available for analytics.

We have successfully implemented this scenario in various projects. Keep in mind the maximum execution limit of 300 seconds when implementing data processing jobs with AWS Lambda.

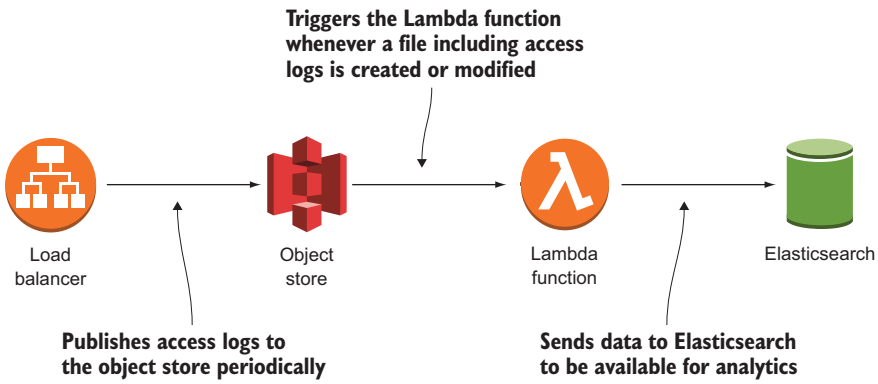


Figure 7.23 Processing access logs from a load balancer with AWS Lambda

7.4.5 Use case: IoT back end

The AWS IoT service provides building blocks needed to communicate with various devices (things) and build event-driven applications. Figure 7.24 shows an example. Each thing publishes sensor data to a message broker. A rule filters the relevant messages and triggers a Lambda function. The Lambda function processes the event and decides what steps are needed based on business logic you provide.

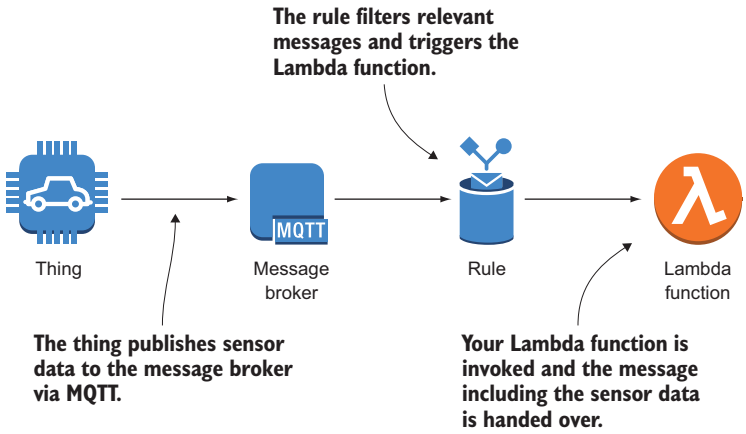


Figure 7.24 Processing access logs from a load balancer with AWS Lambda

We built a proof-of-concept for collecting sensor data and publishing metrics to a dashboard with AWS IoT and AWS Lambda, for example.

We have gone through three possible use cases for AWS Lambda, but we haven't covered all of them. AWS Lambda is integrated with many other services as well. If you want to learn much more about AWS Lambda, we recommend the following books:

- *AWS Lambda in Action* by Danilo Poccia (Manning, 2016) is an example-driven tutorial that teaches how to build applications using an event-based approach on the back end.
- *Serverless Architectures on AWS* by Peter Sbarski (Manning, 2017) teaches how to build, secure, and manage serverless architectures that can power the most demanding web and mobile apps.

Summary

- AWS Lambda allows you to run your Java, Node.js, C#, Python, and Go code within a fully managed, highly available, and scalable environment.
- The Management Console and blueprints offered by AWS help you to get started quickly.
- By using a schedule expression, you can trigger a Lambda function periodically. This is comparable to triggering a script with the help of a cron job.

- The Serverless Application Model (SAM) enables you to deploy a Lambda function in an automated way with AWS CloudFormation.
- There are many event sources for using Lambda functions in an event-driven way. For example, you can subscribe to events triggered by CloudTrail for every request you send to the AWS API.
- The most important limitation of a Lambda function is the maximum duration of 300 seconds per invocation.